

Deep learning for the Sun

John A Armstrong, Christopher MJ Osborne and Lyndsay Fletcher examine how neural networks can be used to explore the nature and location of solar activity.

Machine learning is the process of using statistical techniques to give computers the ability to learn how to perform a specific task without explicit programming. What this means is that, rather than the user having to account for every edge case and boundary condition, the computer works this out via a complex optimization problem. The endgame of a machine-learning algorithm is the development of a data-driven model; given sufficient diversity in the dataset and a clear enough problem to learn, the algorithm can learn about the high-dimensional space spanned by the data. This is equivalent to fitting the function:

$$y=f(x;\theta) \quad (1)$$

where y is the output to achieve, x is the input data to learn from, f is the mapping to learn, and θ are the parameters to be optimized in the system (also known as the learnable parameters). If the data is representative of the problem, there will be examples of almost all cases to model, with the assumption that any cases that the algorithm has not encountered before can be interpolated to by the learned model.

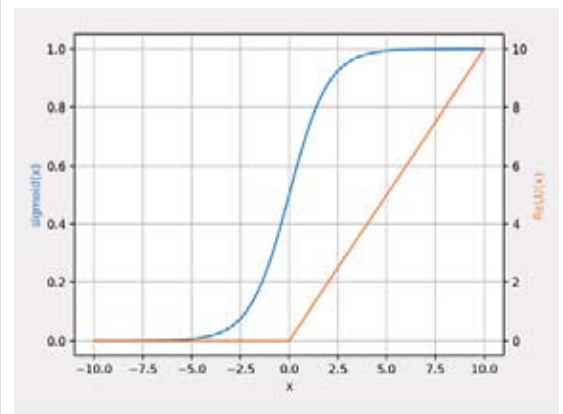
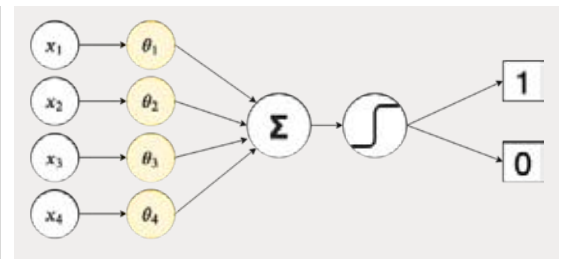
Machine-learning techniques are becoming increasingly important across space science and science as a whole because of three main factors.

First, the amount of data being collected by astronomical instruments has increased exponentially over the past two decades with solar physicists about to enter the petabyte (PB; 1 PB=1000TB) age of data with the Daniel K Inouye Solar Telescope (DKIST; Elmore *et al.* 2014) seeing first light this year in Maui, Hawaii, and expected to produce approximately 10PB of data per year. Other astronomers who have already entered this era will make leaps towards the exabyte (EB; 1 EB=1000PB) age with the Vera C Rubin Observatory Legacy Survey of Space and Time (LSST). This increase in data poses a fundamental challenge: how to process all of this data in a feasible amount of time? Suddenly, simple cataloguing tasks become near impossible, and data analysis on a substantial amount of data seems like a pipe dream. This is a problem that can (and should) be solved using machine learning. These learning algorithms are capable of classification tasks and can be used in conjunction with observation plans to determine what lies within a field of view. These methods have been shown to be able to run in real time in solar (Armstrong & Fletcher 2019) and galactic (Dai & Tong 2018) contexts. Classification methods do rely on some of the data being classified by hand, but it is possible for deep active learning (Wang *et al.* 2017) to be used to get the greatest benefit from the large amount of data that we are receiving.

Second, machine-learning techniques are useful in regression problems and can perform data analysis on a large dataset faster than traditional statistical techniques. One example comes from Asensio Ramos *et al.* (2018), who taught a neural network how to perform multi-object multi-frame blind deconvolution (MOMFBD, van Noort *et al.* 2005), which is used for data reduction of ground-based

1 (Top) Rosenblatt's perceptron. The data x_i are the inputs to the perceptron that are then combined with the weights/learnable parameters θ_i via the vector dot product Σ . This result is then passed to the step function that determines whether or not the neuron "fires", i.e. whether the output is 1 or 0.

2 (Bottom) A comparison of the sigmoid and rectified linear unit nonlinearities.



solar data. A dataset that took days or weeks to be reduced "by hand" can be done in seconds. Another example is the use of invertible neural networks (described below) to infer properties of a solar flare plasma from observations, a feat that had not been accomplished until Osborne *et al.* (2019).

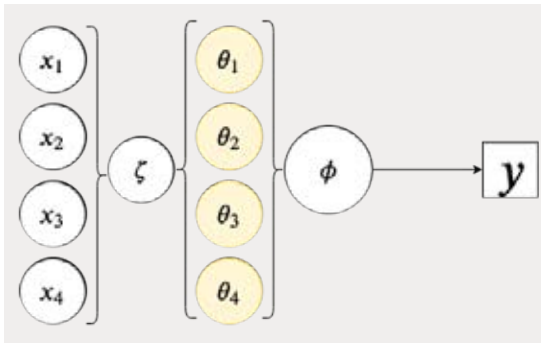
Third, these techniques can be used to uncover hidden patterns and correlations in a dataset. This is important for forecasting where there are many data sources; these techniques can indicate the most important for predictive models. An example of this is using principal component analysis to rank the importance of the features in a dataset and using only those that contribute the most to variations within the data (other features are assumed to be redundant).

At the core of modern machine learning is deep learning, which uses deep neural networks (DNNs) to learn how to do the task at hand. DNNs have been shown (Cybenko 1989, Lu *et al.* 2017) to be able to approximate arbitrary well-defined functions through the stacking of nonlinearities leading to a high-dimensional optimization problem (universal function approximation theorem). There are two major steps in implementing deep learning for research: construction of the DNN and training of the DNN. To understand how this works, we must go back to the conception of machine learning: Frank Rosenblatt's perceptron (Rosenblatt 1951).

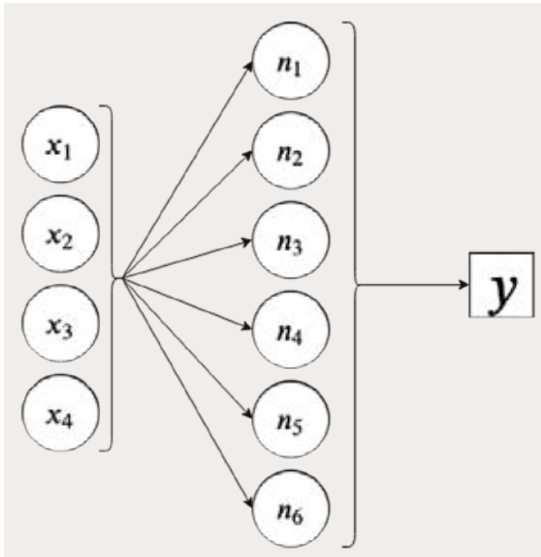
Rosenblatt's perceptron

Rosenblatt's perceptron (hereon referred to as simply a perceptron) is a simple set-up modelled on a single neuron in the brain: there are many inputs with varying electrical signals to a neuron in the brain that are summed together, and if this value is larger than a threshold, the neuron will fire. Within the scope of the perceptron, this translates to a collection of inputs and a so-called "weight vector" that contains the parameters to be learned by the system. The vector inner product is then computed between the input dataset and the weight vector to provide an integrated

"Supervised learning is when the dataset we are trying to learn from is labelled"



3 (Top) The generalized node where ζ is the arbitrary linear function to combine the input x , and weights θ_i before being passed to the activation function ϕ .



4 (Bottom) Example of a neural network with a single hidden layer where the input data is passed to each of the nodes, n_i , independently, where each node is represented as in figure 3 before being passed out and combined into the output.

signal, which is then passed to a step function that determines whether or not the neuron fires.

The perceptron is then trained in what is known as a supervised manner. Supervised learning is when the dataset we are trying to learn from is labelled. In the perceptron example, each input fed to the network for training has a 1 or a 0 attached to it. This is then trained through the “tweaking” of the learnable parameters: an input is passed through the perceptron and the result is calculated. If the result is correct, there is no change to the learnable parameters; if, however, the result is incorrect, then the parameters are tweaked by a small increment. This is done iteratively over the dataset until the highest accuracy is found. A more statistically rigorous and automated training procedure was not introduced until almost 40 years later.

The perceptron is only useful for binary classification, and even then it struggles. To improve performance and versatility of this model, the nonlinearity function is changed. In figure 1, the nonlinearity is simply a step function, meaning that it has only two possible outcomes. This is not useful if there are multiple classes to identify or a continuous problem to learn. It turns out that this step function can be replaced with any nonlinear function and still be capable of learning – this nonlinear function is hereon referred to as an activation function (calling back to the biological anecdote of firing a neuron).

Two typical choices for a replacement of nonlinearity are the sigmoid function and the rectified linear unit (ReLU) function (shown in figure 2).

$$\phi(z) = \frac{1}{1 + e^{-z}} \quad (2)$$

Both the sigmoid, equation (2) and the ReLU, equation (3), allow for regression problems to be learned when replacing the step function and each have their advantages and disadvantages.

$$\phi(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases} \quad (3)$$

The output of a ReLU is sparse and is typically used in the

deepest of DNNs because of this property. Cybenko (1989) proved the universal function approximation theorem for sigmoids, while it took until nearly 30 years later to prove it for ReLUs (Lu et al. 2017). These are just two examples of a wide range of activation functions that can be exploited with deep learning. The key point here is that it should not matter which activation function is used, the function will still be learned, but some activation functions may work better for certain datasets.

Now that the nonlinear part of the perceptron has been generalized (and is now referred to as a node), the linear transformation used to combine the input with the learnable parameters is examined. In the perceptron, the linear function is simply the vector inner product between the inputs and the weights. This means that the number of parameters that the algorithm learns scales linearly with the number of inputs. This gets messy in the context of images, because each pixel must be treated as a separate input, quickly leading to an optimization over millions of parameters.

To combat this, we change the linear function to a convolution and introduce the concept of the receptive field. A convolutional kernel (an initialized matrix with predefined size) is defined to convolve with our input and the elements of this kernel matrix are the learnable parameters for this node. This convolution between input and kernel produces what is known as a feature map. The idea is that rather than having a linear relationship between number of inputs and the number of learnable parameters, each node has a singular convolution kernel that is convolved over the whole input in a sliding window manner (this is known as weight sharing). The weight-sharing drastically reduces the number of learnable parameters per node and also incorporates interesting properties beneficial when working with images. Neighbouring pixels in an image are typically strongly correlated, with the correlation dropping as a function of Euclidean distance from the pixel. This important property is understood by the convolution function and is something that influences the size of the kernel chosen. Also, the convolution function, by construction, deals with shift-invariance, meaning that the relative positions of pixels in an image are not learned but rather the geometry of the features.

Now that the perceptron has been generalized to being a single node, many nodes can be combined in parallel and in series to create a neural network.

Combining nodes: width and depth

A node is comprised of a linear transformation ζ followed by a nonlinear activation ϕ and a corresponding set of learnable parameters. This produces a single output value in the case of the vector inner-product linearity, and a feature map in the case of the convolution. Many of these nodes can then be stacked in parallel, creating what is known as a layer. Each node in a layer creates an output dependent on independent families of learnable parameters. This can allow for each feature map to locate different features in the input.

The process of changing the number of nodes in a layer is known as changing the width of the layer. The system set-up now consists of an input sequence mapping to a layer mapping to an output sequence – this is a neural network (NN). This neural network is defined as shallow because there is only one layer between the input and output. While a shallow network can learn simple tasks, it is often prudent to increase the complexity of the representation by using deeper networks. This means adding more layers to the network.

DNNs use more than one layer to map the input to the desired output. The idea behind using multiple layers

is that with each successive layer, the representation of the data becomes more abstract. The inputs to one layer are the outputs from the previous layer. Rather than having the activations from each neuron in a layer combined to give an output, if they are passed as input to the next layer like the input dataset is passed to the first layer, then successive layers will perform a node operation on already-transformed data. This allows the NN to learn about the transforms of the transform, in a similar way that the second derivative of a function informs us about the nature of the first derivative. In essence, adding more layers allows the network to learn a hierarchical, abstract representation of the data, with earlier layers learning low-level information and later layers learning high-level information. This will be discussed more in the next section.

Typical NN architectures

There are two typical NN architectures: fully connected networks (FCNs) and convolutional neural networks (CNNs). FCNs consist of layers of nodes in which every node in one layer is connected to every node in the proceeding layer (shown in figure 5). Each connection between nodes is a set of learnable parameters where the linearity is the vector inner product, which leads to the learnable parameters being a dense matrix of numbers, as discussed above. These kinds of networks are typically useful in small-scale problems because the linear increase in the number of learnable parameters needed takes a huge computational toll in larger problems. This is particularly apparent for image data.

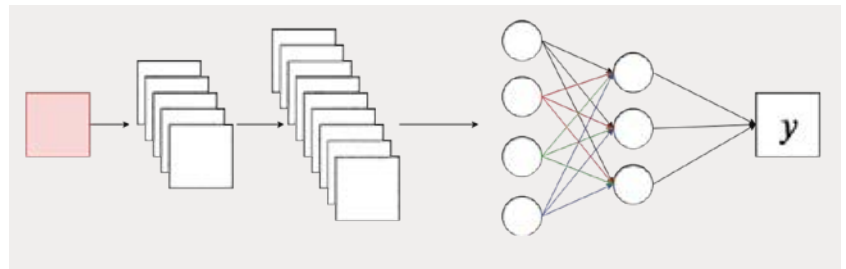
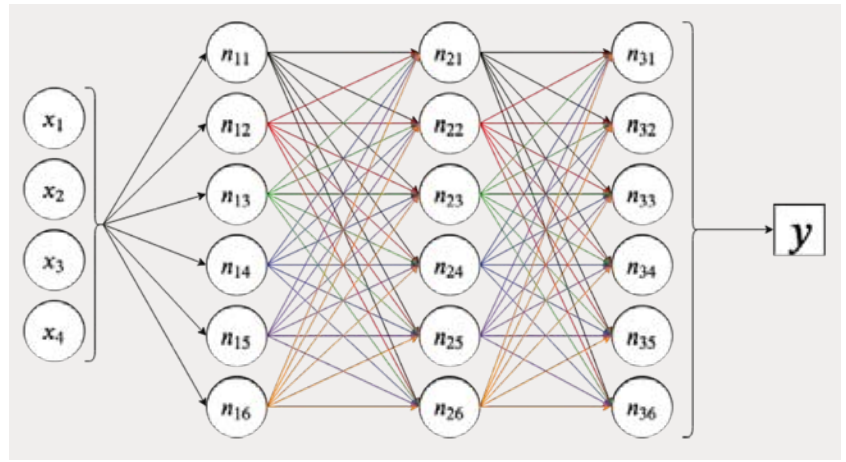
Consider a megapixel image. In an FCN, every pixel is treated as a separate input to the network, meaning that the input layer would have a width $\mathcal{O}(10^6)$ and given that the first hidden layer has width of N , then there are going to be $\mathcal{O}(10^6 \times N)$ connections. This quickly becomes unfeasible.

So how do NNs deal with image data? Lecun *et al.* (1998) has the answer: the inner product linearity is replaced with a convolutional operator (discussed above) to create a CNN. CNNs are based on the visual cortices of animals. In loose terms, the visual cortex of an animal is a series of interconnected neurons that maps an image at the eye to an understanding of what is in the image at the brain, with specific electrical signals being passed between each layer of neurons depending on the features in the image. This is achieved in a hierarchical manner, meaning that the earlier layers detect coarse features such as colour and gradients, whereas the later layers detect fine features such as a word or a nose. Each layer of the system itself does not pick out specific features of objects, but rather an abstract feature representing a property of the image (different objects to be learned will have specific values of these abstract features). CNNs achieve this through downsampling of images as they are passed through the network. The goal is that in downsampling the image, each pixel will represent more information, leading to more abstract features being extracted deeper in the network.

Training a DNN

Now, consider a DNN that contains M layers each with N nodes and each node performs a linear/nonlinear transformation on a given set of data using a set of learnable parameters $\{\theta_j\}_{MN}$. This defines a system with a large number of unknown parameters that we want to be able to map an input to a correct output for a certain task. This is where the learning comes in. By learning though, we really mean optimization. Very, very high-dimensional optimization.

Initially, training one of these networks required the modification of the learnable parameters by hand. A



5 (Top) A fully connected network. All of the nodes of one layer are connected to all of the nodes in the proceeding layer. This is a typical set-up that utilizes the vector inner product as the linearity in the nodes. Each connection between nodes is a set of learnable parameters in this system.

6 (Bottom) A convolutional neural network. The first two layers indicate the feature extraction layers with an increasing number of feature maps. These are then passed to a set of fully connected layers to map to an output. The red square here is the input image to the network.

predicted output would be compared with the true output using a metric and then the learnable parameters would be tuned manually to try to improve this score. This feedback loop would continue until some form of convergence was satisfied. This approach was fine for simple networks modelling simple problems that had up to 10s of nodes in a single layer. However, for more complex problems, these simple networks do not house the complexity needed. Network size increases, culminating in the hand-optimized training of a network becoming unfeasible. This is a problem that really put a damper on the field of neural networks, because they seemed very limited in what they could actually learn.

Automated training for a DNN utilizes a very powerful tool known as backpropagation. Traditional NN are trained in a supervised manner: the input is passed through the network and a result is obtained (feed-forward). The calculated output is then compared with the correct output using a metric known as a loss function. The value of the loss function is then what matters for how a network learns. The loss is a function of the input and output data (which are both fixed) and the learnable parameters (which are variable). As a result, gradients of this loss function with respect to the learnable parameters can be calculated and a gradient descent method is used to update the parameters. The gradient descent method of choice for training DNNs is known as stochastic gradient descent (SGD) and will update a parameter according to the following:

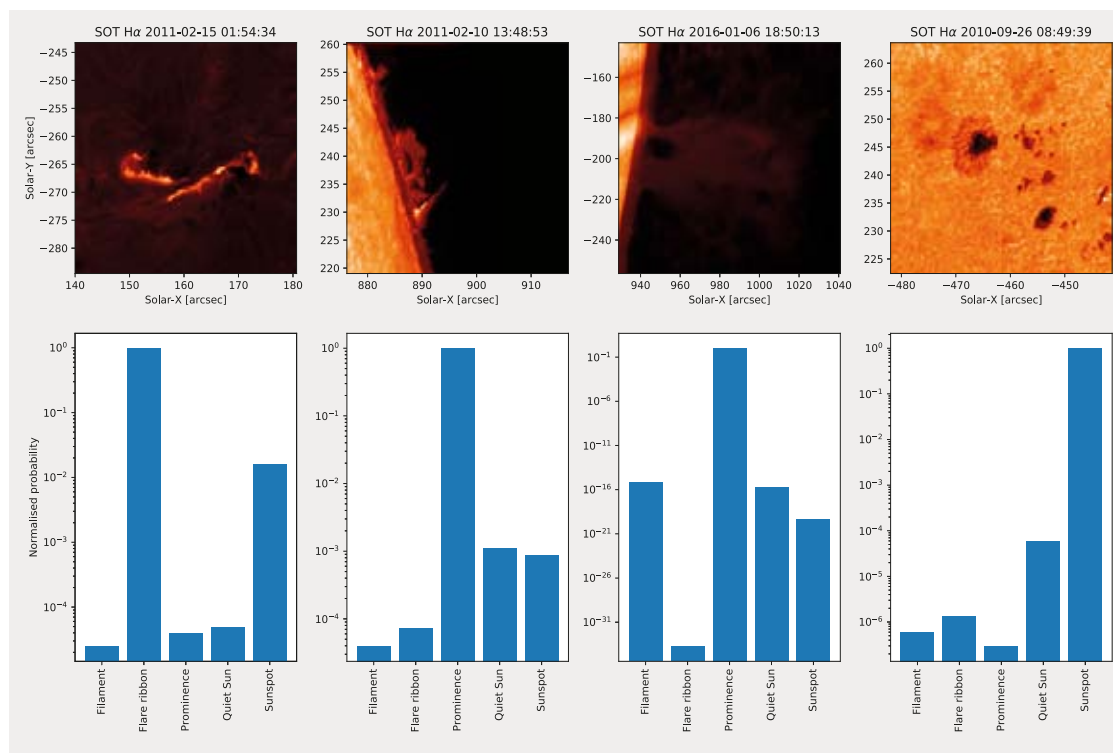
$$\theta^{t+1} = \theta^t + \eta \nabla L(x, y; \theta^t) \quad (4)$$

That is, each of the learnable parameters in the system are updated via equation (4). The difficulty comes now in the form of calculating these gradients. Luckily, this has been made easier by construction of the network. Each of the outputs of the network can be expressed as the composition of many functions applied to the input:

$$\tilde{y} = (\phi_M \circ \zeta_M \circ \phi_{M-1} \circ \zeta_{M-1} \circ \dots \circ \phi_1 \circ \zeta_1)(x) \quad (5)$$

where \circ denotes the composition of two functions, i.e. $(g \circ f)(x) = g(f(x))$. Equation (5) follows the notation introduced in figure 3. \tilde{y} is our estimate of the true output y . The gradient for the learnable parameter θ_i^t ($i \in M$) can then be found using the chain rule:

$$\frac{\partial L}{\partial \theta_i^t} = \frac{\partial L}{\partial o_i^t} \frac{\partial o_i^t}{\partial \theta_i^t} \quad (6)$$



7 Example of a convolutional neural network classifier in solar physics from Armstrong & Fletcher (2019). The four images above show flare ribbons (left), prominences (middle) and sunspots (right). None of these images were used to train the network and the histograms below show a discrete probability distribution for each image. The network classifies these images correctly.

where o_l^i is the output of layer M . It can be shown that the change of the loss with respect to the output is dependent on the subsequent layers that layer l serves as input to. This is where backpropagation comes in. In the simplest sense, for a learnable parameter in the output layer, the change in loss depends only on the estimate of the output. Working backwards through the network, the next layer down will depend on the change of the loss from the output estimator and the output of that layer, and so on and so forth. Logically, working out gradients backwards and updating them as the network is traversed backwards thus provides a tool for optimization. Using the properties of equations (5) and (6) in combination with equation (4) will lead to a system that adjusts the learnable parameters based on how the outputs throughout the network change. This is what we define as learning.

Optimization via backpropagation and gradient descent is not magic; it depends on human-picked parameters known as hyperparameters. The hyperparameters are not things that the system could pick sensible values for. In the most basic of training set-ups, we have three hyperparameters: learning rate, number of epochs and batch size.

The learning rate is the η term in the SGD equation (4) and is the same as a step size used in other gradient descent techniques. In backpropagation, the learning rate determines how big (or small) a step to take when traversing the space of learnable parameters on the loss function. This can be vital; a learning rate too large can lead to a network that will never converge because it will continually hop over the troughs of minimal loss that it is looking for. On the other hand, a learning rate too small will result in the network falling into the first local minimum that it encounters and never being able to escape. The idea is to find a medium place between these two extremes and hope that the algorithm can land in a local minimum that is an area of minimal loss.

The number of epochs is the number of times that backpropagation will be performed on the model. This is an important number to experiment with, because if it is too small, the network will not have enough time to learn and generalize, whereas if it is too large, the network will overfit. The typical way of testing for the correct number of epochs without overfitting is using a validation dataset. This is a small subset of the training dataset (typically

10–20%) that is not used in backpropagation. Instead, the validation dataset is passed through the network at the end of an epoch; calculating the loss for the validation set then serves as a good indicator of the learning of the system. Ideally, for a NN, the dataset should be balanced between all classes and should contain different examples of those classes. This gives the network the largest possibility of learning the function. In this case, the validation dataset can be drawn randomly from the training dataset. The measure of the loss over the validation dataset can then be seen as a direct measure of the algorithm's learning: it can successfully apply the function to data that it has never seen before but is similar to the training dataset.

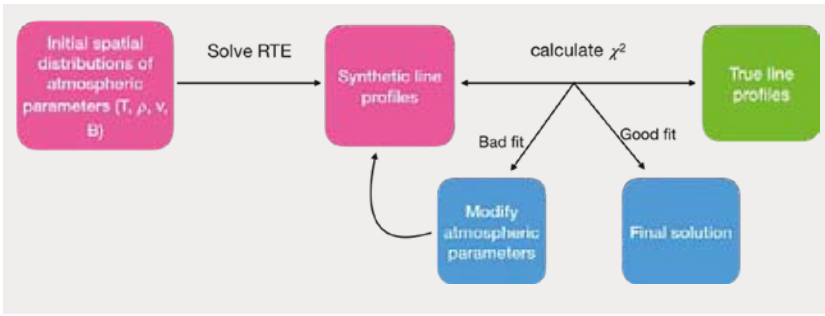
The third hyperparameter discussed here is the batch size. Batch size is a funny thing and its importance is not as obvious as the first two. The concept of batch size comes from SGD itself. What makes SGD “stochastic” is that a group of data is taken (a batch) and their weights updated not based on their individual losses, but instead on the means of their losses. This has a computational advantage because fewer gradients need to be calculated, but it also works in a learning sense. The averaging of the losses over a batch means that the parameter updates will be “fuzzy”; rather than pulling the parameters towards a solution for one input, they are moved generally in the direction of a solution that works the best for most of the inputs. Batch size is limited by computing power – there are only so many inputs that the hardware can compute in parallel at one time – but also by optimization – it is better to have multiple batches per epoch to make multiple parameter updates that will all contribute positively to the optimization.

A CNN in solar physics

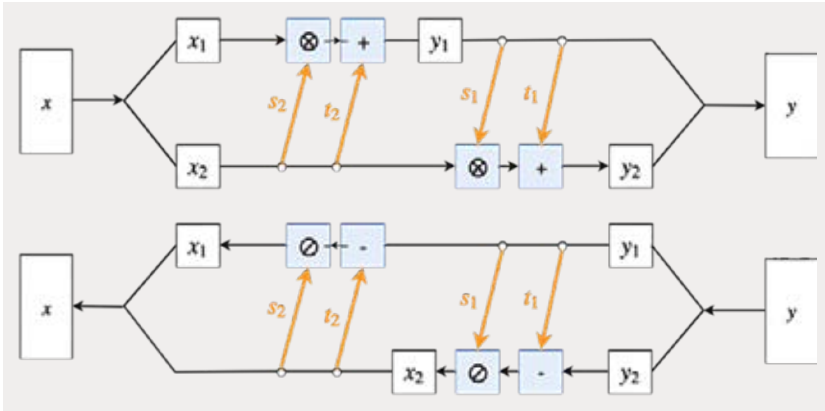
An example of a network comes from Armstrong & Fletcher (2019) that uses a deep CNN to classify high-resolution images of the solar chromosphere. This network was developed to demonstrate the value of deep-learning techniques for cataloguing in solar physics and will lead to an eventual application in transfer learning, which is the process of using a trained DNN to help a new DNN learn the task at hand.

The network is similar in structure to that shown in figure 6: there are 10 feature extraction layers followed by

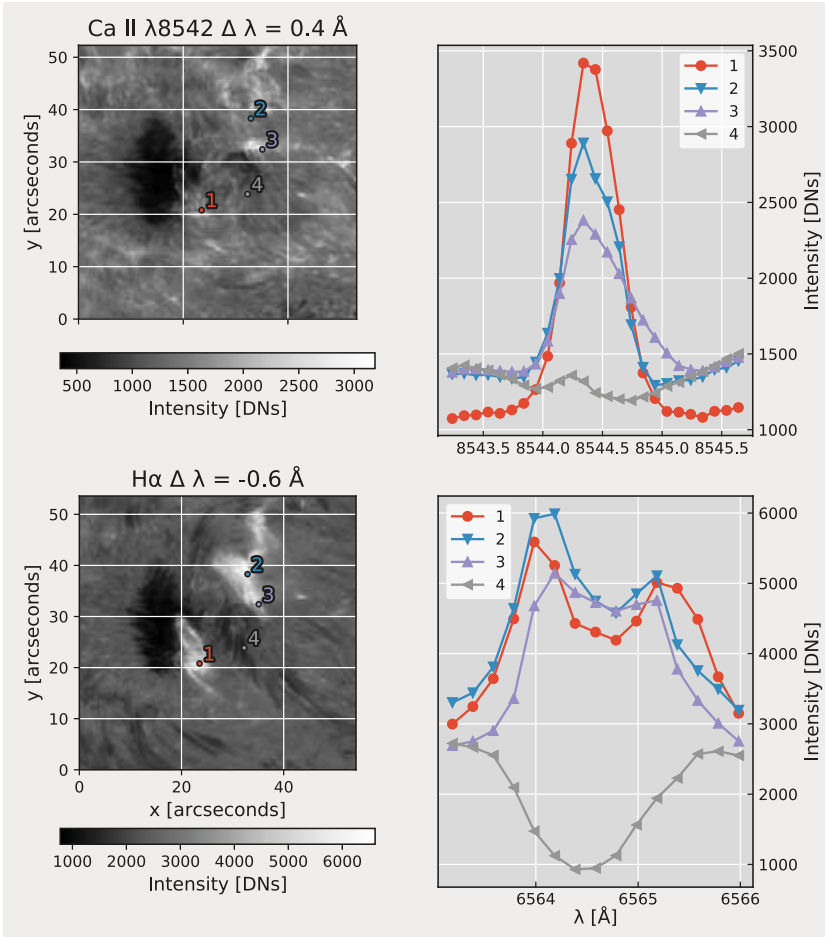
“There are two typical NN architectures: fully connected networks and convolutional neural networks”



8 A schematic diagram for how inversions work, starting with an initial atmosphere and solve the radiative transfer equation to generate synthetic line profiles. These are compared with the observed line profiles using a χ^2 metric. If the χ^2 is below a certain threshold, the atmosphere is accepted as being the atmosphere responsible for the observables. If it is too large, then atmospheric parameters are modified using gradient descent and recompute the line profiles. This process is repeated until convergence.



9 Schematic of the affine-coupling layer described by equation (7). The input is passed to the network where it undergoes an affine transformation. The inverse of this layer then comes from simply reversing the direction of information flow. The learnable parameters from these layers come from the $\{s_i, t_i\}_{i=1,2}$ functions whose inverses are never calculated therefore are chosen to be fully connected networks.



three fully connected layers that map the deepest feature maps to the class representing the features within them. The features that we chose to learn are filaments, prominences, sunspots, flare ribbons and “quiet” (the absence of the other four features).

The network performs well with 99.7% accuracy on the validation dataset after training and has good generalization properties to other wavelengths. An interactive example of how to use this network is available at github.com/rhero12/Slic. This network can classify more than 1000 images in less than a second, and an example of classification of images it has not been trained on are shown in figure 7.

Parameter estimation using deep learning

Parameter estimation is a ubiquitous method in astronomy for determining physical parameters from observables. In the lower solar atmosphere this is typically done using inversion techniques where spectral lines are optically thick – i.e. extinction along the line-of-sight is imperative to model the line profiles observed. Inversion techniques for optically thick spectral lines classically take the form of forward modelling: starting with some standard solar atmosphere that will have height distributions for the atmospheric parameters (e.g. density, temperature, magnetic field and velocity), solve the equation of radiative transfer to get the spectrum as a function of wavelength. The calculated spectrum is compared with the observables and a χ^2 statistic is calculated. If χ^2 is less than some threshold, the atmospheric parameters have produced the observables. If not, a gradient descent technique is used to update the parameters at fixed nodes in height. The process is then repeated until convergence, as shown in figure 8.

A major issue is that the inversion process (that is, the mapping from the observables to the atmospheric parameters) is not well defined, meaning that traditional deep-learning techniques lack the ability to learn the problem. That is, the function from atmospheric parameters to observables is deterministic but the inverse is not. There is some physical information lost when computing the observables that makes the recovery process of the atmospheric parameters ambiguous. This could be loss of information about where emitted light of different wavelengths originate, because optically thick lines form in a region of the atmosphere where opacity effects are important. Thus, multiple configurations of the atmospheric parameters can produce the same observables. To resolve this and to recover a one-to-one mapping, an algorithm that can learn not only the forward radiative transfer problem but also the information that is lost in the forward process is investigated. The information lost is referred to as coming from a latent space that is a distribution containing information that could be lost.

Traditional network architectures such as FCNs or CNNs rely on large matrix operations; as a result, the inversion of these networks may not be possible because the transformation matrices could be singular, i.e. matrices without inverses. Therefore, the architecture used is known as invertible neural networks (INNs; Ardizzone *et al.* 2018). They are constructed of affine-coupling layers that transform the input to the output using affine transformations that have an easy-to-track inverse, i.e.

10 (Left) Observations of the 6 September 2014 solar flare in Hα (bottom) and Ca II λ8542 (top). This is a snapshot from just after the onset of the flare as seen by the bright flare ribbons within the field-of-view where we have chosen an image in the red wing of calcium and the blue wing of Hα. Points 1–4 correspond to the spectra in the right-hand column.

$$\begin{aligned}
y_1 &= x_1 \otimes \exp(s_2(x_2)) + t_2(x_2) \\
y_2 &= x_2 \otimes \exp(s_1(y_1)) + t_1(y_1) \\
x_2 &= (y_2 - t_1(y_1)) \otimes \exp(-s_1(y_1)) \\
x_1 &= (y_1 - t_2(x_2)) \otimes \exp(-s_2(x_2))
\end{aligned} \quad (7)$$

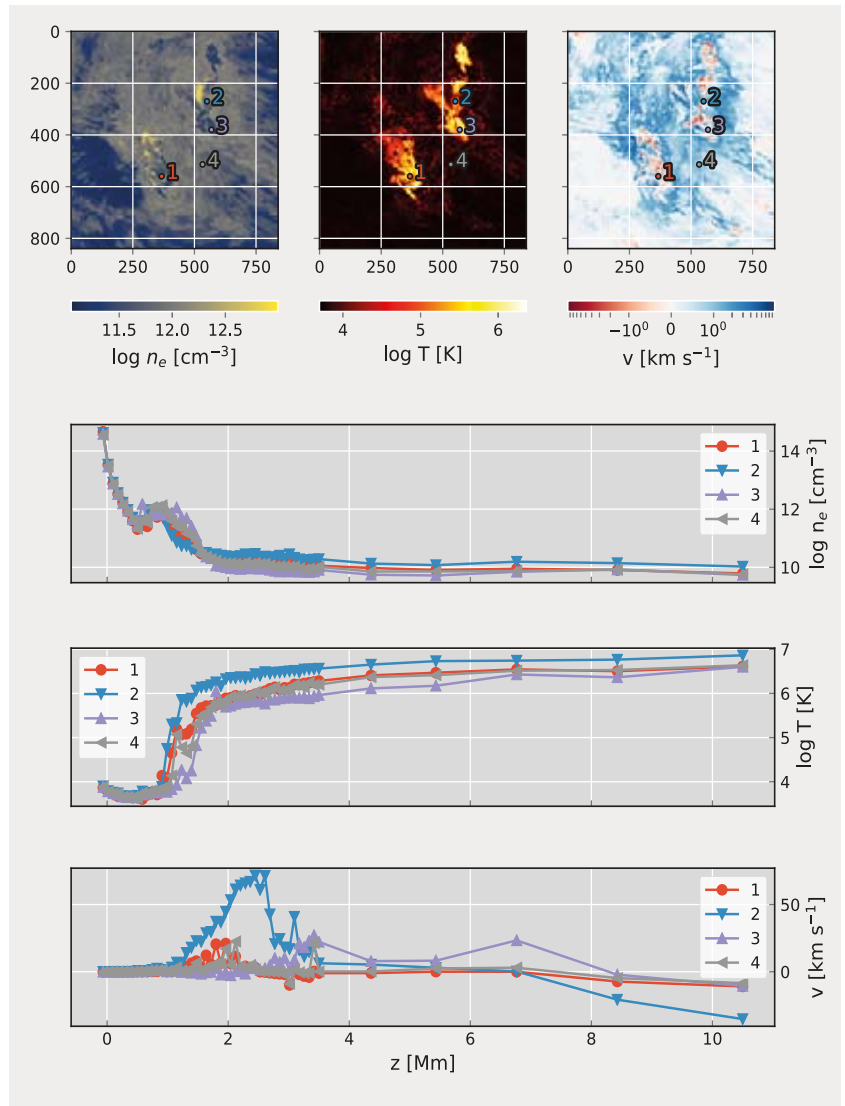
The inverse of an affine transform is trivial as can be seen in equation (7); the important point to note is that in finding the inverse of the layer we do not need to find the inverses of the set of functions $\{s_i, t_i\}_{i=1,2}$. They can be arbitrarily complex (and not necessarily invertible). This is how the learnable parameters of the system are defined; the functions are chosen to be FCNs that will learn the optimal affine transform to produce the desired output while also providing the inverse for free.

Multiple affine-coupling layers are stacked together into an invertible neural network. This network is trained on simulations of solar flares and their generated profiles of two spectral lines – H α and CaII 8542 – using three atmospheric parameters: electron number density, electron temperature and bulk velocity flow of the plasma. This is then applied to observations of a solar flare from 6 September 2014 using data from the Swedish Solar Telescope's CRisp Imaging SpectroPolarimeter (CRISP). These observations are narrow-band imaging spectroscopy in both of the lines mentioned above. The time range for the observations is 15:00–18:30 UTC so there are observations from the pre-flare and the entire lifetime of the post-flare ribbons in both of the lines (Osborne *et al.* 2019). Every image is inverted to investigate the role of dynamics in the flaring chromosphere and the spectral lines that these atmospheres produce. The inversion of one image takes approximately 30 minutes, which is orders of magnitude faster than classical inversion techniques.

The asymmetries and shapes of the observed spectral lines provide a lot of information about the dynamics of the plasma. For instance, the lines considered should be symmetric about the line core in a static atmosphere; given that the core of the lines (as can be seen in figure 9) are Doppler-shifted, the atmosphere cannot be static. Furthermore, this is attributed to chromospheric evaporation/condensation that are the bulk expansion flows that occur in the rapidly heated flare chromosphere. Mapping these asymmetries back to atmospheric parameters is made complicated by absorption and emission in the moving plasma. For example, the blue asymmetries observed in the wings of the H α lines in figure 10 could arise either from increased emission in upflowing plasma or increased absorption in downflowing plasma. These are the kinds of ambiguities yet to be resolved. Kuridze *et al.* (2015) find that the formation region of the H α wings occur below 0.95 Mm in the atmosphere and therefore studying the inversions there may help to resolve the ambiguities in the line profiles. As can be seen in figure 11, the inversion suggests that just after flare onset there are regions of both increased emitting upflows and increased absorbing downflows. There is much more exploration to be done with this data that may lead to substantial insight into the flaring chromospheric plasma.

Conclusion

Machine-learning techniques can be important tools for space scientists in data exploration/cataloguing, function approximation and uncovering features in the data not explicitly obvious – especially as we enter the age of big data. Machine learning should not be viewed as a “black box” any more than any other optimization technique is, and should be used accordingly. Hopefully, this has been a good illustration of what machine learning and, more specifically, deep learning, is capable of and how it can complement other areas of data analysis. ●



11 The top row shows the inversions for the upper limit of the region where the wings of H α form ($z = 0.91$ Mm) according to Kuridze *et al.* (2015). The line profiles from figure 9 of H α on the flare ribbons show strong blue asymmetries in the wings that we can see from the inversions are the result of both downflows and upflows. This means that, after the onset of the flare, there are areas where material emitting H α is moving upwards and other areas where material that absorbs H α is moving downwards. This is demonstrated in the three lower rows that are the atmospheric parameter profiles with height for each of the points 1–4.

AUTHORS

John A Armstrong is a PhD student at the University of Glasgow, whose passion for astrophysics arose from enjoying mathematics, science fiction and the properties of light
Christopher MJ Osborne is a PhD student at the University of Glasgow, who investigates solar energy transport by encouraging – and occasionally torturing – computers
Prof. Lyndsay Fletcher is professor of astrophysics at the University of Glasgow, being educated in machine learning by John and Chris



ACKNOWLEDGEMENTS

JAA thanks A Peat for constructive feedback on this article. This research was made possible by grants from the UKRI's STFC

REFERENCES

Ardiszone L *et al.* 2018 Analyzing

inverse problems with invertible neural networks *ICLR* 2019
Armstrong JA & Fletcher L 2019 *Solar Physics* 294(6) 80
Asensio Ramos A *et al.* 2018 *Astron. Astrophys.* 620 A73
Cybenko G 1989 *Mathematics of Control Signals and Systems* 2 303
Dai J-M & Tong J 2019 *Astrophysics and Space Science* 364 55
Elmore DF *et al.* 2014 *Proc. SPIE* 9147 914707
Kuridze D *et al.* 2015 *Astrophys. J.* 813(2) 125
LeCun Y *et al.* 1998 *Proc. IEEE* 86(11) 2278
Lu Z *et al.* 2017 The expressive power of neural networks: a view from the width *Proc. NIPS* 2017
Osborne CMJ *et al.* 2019 *Astrophys. J.* 873(2) 128
Rosenblatt F 1958 *Psychological Review* 65(6) 386
Van Noort M *et al.* 2005 *Solar Physics* 228(1–2) 191
Wang K *et al.* 2017 *IEEE Transactions on Circuits and Systems for Video Technology* 27(12) 2591